Design of Access Control Policy Checker (ACPC)

A thesis submitted in partial fulfillment
of the requirements for the degree of

Master of Technology

in

Computer Science and Engineering
Specialization: Information Security

by

Suraj Sharma

under the guidance of

Dr. Sanjay Kumar Jena

Department of Computer Science and Engineering
National Institute of Technology Rourkela
Rourkela-769 008, Orissa, India

May 2009

To my parents

Department of Computer Science and Engineering
National Institute of Technology Rourkela
Rourkela-769 008, Orissa, India.

Certificate

This is to certify that the work in the thesis entitled ***Design of Access Control Policy Checker (ACPC)*** submitted by ***Mr.Suraj Sharma*** in partial fulfillment of the requirements for the award of the degree of Master of Technology in Computer Science and Engineering during the session 2008–2009 in the department of Computer Science and Engineering, National Institute of Technology Rourkela is an authentic work carried out by him under my supervision and guidance.

To the best of my knowledge, the matter embodied in the thesis has not been submitted to any other University/Institute for the award of any Degree or Diploma.

Dr. Sanjay Kumar Jena
Professor
Dept. of Computer Science & Engineering
National Institute of Technology
Rourkela-769008 Orissa (India)

Place: NIT Rourkela
Date: 12 May 2009

Acknowledgment

My first thanks are to the Almighty God, without whose blessings I wouldn't have been writing this "acknowledgments".

I then would like to express my heartfelt thanks to my guide, **Dr.Sanjay Kumar Jena**, for his guidance, support, and encouragement during the course of my master study at the National Institute of Technology, Rourkela. I am especially indebted to him for teaching me both research and writing skills, which have been proven beneficial for my current research and future career. Without his endless efforts, knowledge, patience, and answers to my numerous questions, this research would have never been possible.The experimental methods and results presented in this thesis have been influenced by him in one way or the other. It has been a great honor and pleasure for me to do research under Dr. Sanjay Kumar Jena's supervision.

I am very much indebted to **Dr.B.Majhi**, Head of the Department, Computer Science engineering, National Institute of Technology, Rourkela for his support during my work.

I am grateful to **Dr.A.K.Turuk** for teaching me the right way to present the motivation of my thesis. His insightful feedback helped me improve the presentation of the thesis in many ways. I am also thankful to Dr.S.K.Rath, Dr.D.P.Mohapatra, Dr.R.Baliarsingh, Dr.P.M.Khilar, Sr.Lectuter Bibhudatta Sahoo, Lecturer Pankaj Sa, Lecturer K. Sathyababu, for giving encouragement during my thesis work.

I thank all the members of the Department of Computer Science and Engineering, and the Institute, who helped me by providing the necessary resources, and in various other ways, in the completion of my work.

Finally, I thank my parents and all my family member for their unlimited support and strength. Without their dedication and dependability, I could not have pursued my M.Tech. degree at the National Institute of Technology Rourkela.

Suraj Sharma

Abstract

Any multiuser system has to enforce access control for protecting its resources from unauthorized access or damage. One way for specifying access control is in a separate policy specification language. An access control system maintains a repository of policies, receives access requests, consults the policy and returns a response specifying whether the request was permitted or denied. However, it is challenging to specify a correct access control policy and so, it is common for the security of a system to be compromised because of the incorrect specification of these policies. There are many ways in which a policy can be checked for correctness like, formal verification, analysis and testing. In this thesis, a systematic and automatic tool for policy testing is provided. Testing a policy involves formulating requests that represent test cases for the policy, evaluating the policy with those test cases (requests) and comparing the responses obtained with actual expected results.

In the approach to policy testing, we conducted the change-impact analysis for generating the requests, and mutation testing for testing the specified policy. The testing framework called ACPC (Access Control Policy Checker), used Margrave tool to perform change-impact analysis for generating requests. We have choosen like previous work [22] an access control specification language, Extensible Access Control Markup Language (XACML).

We conducted experiments using nine policy sets to evaluate the effectiveness by the framework. The experimental result shows that ACPC can effectively generate requests to achieve high structural coverage of policies and outperforms random requests generation in terms of structural coverage and fault-detection capability. We have used nine mutation operators to make the mutant policy for mutation testing. We found the better result by classify these mutation operator in to three classes. We got up to 98% of mutant killed by one class of mutation operator, this results shows that, the classification gives better performance in terms of cost and time.

Contents

List of Abbreviations

XACML	Extensible Access Control Markup Language
XML	Extensible Markup Language
EPAL	Enterprise Privacy Authorization Language
RBAC	Role Based Access Contol
P3P	Platform for Privacy Preferences
W3C	World Wide Web Consortium
PEP	Policy Enforcement Point
PDP	Policy Decision Point
FAM	Flexible Authorization Manager
EDI	Electronic Data Incterchange
ACPC	Access Control Policy Checker
PSTT	Policy Set Target True
PSTF	Policy Set Target False
PTT	Policy Target True
PTF	Policy Target False
RTT	Rule Target True
RTF	Rule Target False
RCT	Rule Condition True
RCF	Rule Condition False
CRE	Change Rule Effect

List of Figures

List of Tables

Chapter 1

Introduction

Introduction

Literature Review

Problem Defination

Objective

Thesis Layout

Chapter 1

Introduction

1.1 Introduction

Any type of system, having different users, need to have a access control system for authorized access and prevention of harm. Access Control System which is specifying separately in the system by a separate policy specification is a solution for that problem. Access Control System contains the different policies, whose work is to receive access requests then it consults to the policy and then returns a response, specifying that the user request is permitted or denied.To implement these access control policies are not an easy task, because of the huge system requirement. For checking the correctness of the policies which is deployed into the system are very difficult. In this thesis, a systematic and automated toll [22] for policy testing is implemented, For test a policy it involve generation of test cases, evaluation of policies with respect to those test cases and at last comparison between the existing approaches.

The advantages of using policy specification languages have led to the development of many specific and generic policy languages. Ponder [1] is an object oriented policy specification language for distributed systems management. Enterprise Privacy Authorization Language (EPAL) [2] is a formal language used to specify fine-grained enterprise privacy policies. Extensible Access Control Markup Language (XACML) [3] is a general purpose policy language and an access request/response language defined using Extensible Markup Language (XML) for managing access to resources. The XACML specification in its XML format en-

ables access policies to be transportable and also inter operable across various access control systems.

As an example, consider the case of firewall policies. Firewalls are one mechanism for securing network resources. It is common for miss configured firewall policies to be causing problems. In examining 37 firewalls in production enterprise networks in 2004, Wool found that all the firewalls were miss configured and vulnerable [4]. In addition, the study states, "The protection that firewalls provide is only as good as the policy they are configured to implement. Analysis of real configuration data shows that corporate firewalls are often enforcing rule sets that violate well established security guidelines". The wide and continued spread of worms such as Blaster and Sapphire, demonstrated that many firewalls were miss configured, because "well-configured firewalls could have easily blocked them". There can be many anomalies and inconsistencies in the policy which make the network resources vulnerable to security attacks. Also, firewall rules are developed over a period of time. New rules are periodically added as more resources with new constraints are added to the network. It is difficult to check for conflicts or overlaps of new rules with existing rules. Similar problem exists in access control policies of enterprises and other systems. An enterprises' security policy is also revised over time as new security requirements are added. Therefore, it is critical to specify access control policies correctly, which however is a challenging problem.

There are various ways in which the quality of the policy can be assured like, formal verification, analysis and testing. Formal verification techniques can verify if a policy satisfies a particular security property [5, 6]. However, a formal representation of the policy is not scalable and properties about a policy do not exist in practice. Analysis of policies can include semantic analysis like performing a change impact analysis between two policies [7]. Testing is one practical way for checking the correctness of a policy specification. Semantic analysis techniques can be used complementary to testing.

Access control is traditionally enforced by directly hard coding into a system. However, this is tedious and becomes difficult for a large system. Also, this makes

it hard to accommodate changes of security requirements in a system. Recently, access control system increasingly separate policy from mechanisms. That is, an access control policy is explicitly specified using certain policy languages. And a system dynamically consults the policy to determine whether an access request should be granted. The advantage of this is that by separating policy from mechanism makes it easier to specify the protection requirements to be enforced on the system independent of the underlying implementation details. Also, when the security requirements on the system change later on, it is possible to easily change the policy without affecting the underlying mechanism implementing it.

There is the existing policy testing techniques. Martin, Xie, and Yu [8] have developed a random test generation tool for XACML policies. The tests (requests) are generated as a set of all combination of attributes found in the policy. The tool represents this attribute as a bit vector and an attribute appears in the request only if the corresponding bit in the vector is set to 1. The number of requests to be generated can be user specified. To achieve adequate coverage, even in a small request set, they modify the random bit setting algorithm to ensure each bit is set at least once. This method, though simple to implement is not ensure that a policy is thoroughly tested.

In our approach to policy testing, we generate policy requests by the Chang-Impact analysis tool margrave and mutation testing method for testing the access control policies[22].

1.2 Literature Review

In this section, we discuss related work to access control models, polices and other techniques that are used for analyzing access control polices[6].

1.2.1 Policies, Models and Mechanisms

Any system implementing access control must consider the three abstractions [9]:

1 Security Policy: This defines high level rules according to which access to resources and data within a system will be granted or denied. An example of a

security policy used at a school could be the TA can assign only internal grades.

2 Security Model: This gives a formal representation of how the access control security policy is implemented in the system. This can be used to give a proof of the properties provided by the system. It can be said that the model bridges the gap in abstraction between policy and mechanism [10]. An example security model is the mandatory access control model, where the level of access of an entity depends on the security clearance level assigned to it like top secret, secret, normal.

3 Security Mechanism: This defines the actual system specific functions that implement the controls imposed by the policy and formally stated in the model. An example security mechanism is access control lists.

1.2.2 Access control models

Access control models are grouped into three main classes: discretionary model, mandatory model and role based model. Our approach to policy testing can be applied to all policies build on any of these models.

Discretionary Policy Model

In discretionary access control [11], a list of authorizations is specified for each subject in the system. The system gives access to a subject by looking up whether a subject has access to an object in the authorizations specified. Different subjects can have different levels of access to one object. In this model, the users have the discretion of granting or revoking privileges to other users. The access matrix model is used for describing discretionary access control. In access matrix, the rows are the subjects in the system and the columns are resources to which a subject's access has to be controlled. The cell intersecting the row and column will specify the access level of the subject to the resource. This matrix model can be implemented as,

- **Authorization table :** Here the authorizations are represented as a table. This is mostly used in databases by creating a table with columns subject, resource and action. Each entry in the table represents an authorization.

- **Access control list :** In an access control list implementation, every column in the access matrix is a list.(i.e) There is a list for each object in the system specifying the subjects that have access to that particular object.

- **Capability list :** In a capability list implementation, every row in the access matrix is a list. (i.e.) there is a list for each subject in the system specifying the different objects that the particular object has access.

Each implementation has its own advantages and disadvantages and a particular implementation is chosen depending on the needs of the specific application. Discretionary policies however are not secure against attacks from the processes invoked by legitimate users that may perform malicious functions on behalf of the user. An example of this vulnerability is a trojan horse program that is executed by a subject like a high level user that reads from one sensitive file and writes to another common file to which a lower level user has read access. Now the low level user will be able to read the contents of the sensitive file.

Mandatory policy model

Mandatory policies classify the subjects and objects within the system into different security clearance levels. The various mandatory policies based on the semantics of the classification are,

- **Secrecy-based mandatory policies :** these control the 'direct and indirect flow of information to the purpose of preventing leakages to unauthorized subjects' [11]. Users can connect to the system at different levels and the two Bell La Pendula principles to be satisfied are: No-read-up and No-write-down. Enforcing this restriction ensures that no information flow exists from one level to another.

- **Integrity based mandatory policy :** The Biba model protects the integrity of a resource. The integrity classification reflects the trustworthiness of the user in modifying the information and for an object it refers to the trustworthiness placed on the data provided by the system. Access control is enforced by the following two principles: No-read-down and no-write-up.

> Enforcing this principle safeguards the integrity by ensuring that objects at a lower level which are less reliable cannot write to levels above it.

Hence, secrecy policies allow the flow of information from lower to higher secrecy classes while integrity policies allow the flow of information from higher to lower integrity classes. So to ensure both secrecy and integrity both the classes must be defined.

Though mandatory policies provide protection against information leakages, they cannot guarantee complete secrecy because they do not offer protection from covert channel communication.

Mandatory and discretionary polices are combined and the Chinese wall policy model is defined. This policy model was proposed to enforce the mandatory control on discretionary policy implementations found in commercial systems. It combines mandatory and discretionary policies. The classification class restricting the information flow here reflects the flow of information between conflicting business classes for an individual consultant. Here, access to data is not constrained by its classification but by what data a subject has already accessed. Though this policy has some limitations of mandatory policies like being rigid in a commercial setting, this is a good example of applying 'dynamic separation of duty constraints present in the real world and has been taken as a reference in building subsequent policies and models' [11].

Other work combining discretionary and mandatory access control include authorization based information flow policies. Also, discretionary policies have been modified for expanding authorizations to support conditions in the policy. Also, authorizations can be extended with temporal constraints.

Another aspect of access control is the administrative policies which specify who is authorized to manage the access rules and decisions. In mandatory, there must be a centralized authority specifying the security class of the objects. In the case of discretionary, there can be different subjects like, centralized, hierarchical, cooperative, ownership and decentralized.

Role based policy model

Role based access control (RBAC) [12, 13] specify access based on what roles the users of the system assume. In [14] gives a novel framework and its supporting tool that generates tests based on change-impact analysis which is the model in this thesis. In this thesis we have used this paper as a main reference.

1.2.3 Policy Specification Languages

The policy testing technique can be applied to other policy specification languages also. Here, we describe some of the common policy specification languages [6].

```
Inst(auth+ | auth-)policyname
Subject domain-scope-expression;
Target domain-scope-expression;
Action domain-scope-expression;
[When constraint-expression]
```

Figure 1.1: Ponder Authorization policy syntax

```
Inst(auth+)policyname
Subject faculty;
Target grades;
Action Assign, View;

//This policy authorizes faculty to assign and view grades.
```

Figure 1.2: Ponder Authorization policy example

Ponder Policy Specification Language

Ponder was developed as part of an academic project at Imperial College in London. Ponder is a declarative object oriented policy specification language. It is more suitable for access control enforcement in distributed and network systems. They separate policy from implementation and enable dynamic management of the policies [6]. The key terms are,

- **Subject :**Subject refers to users or principles or any other automated entity which has a management responsibility.
- **Target :**Target refers to resources or services in the system.

- **Domains :**Domains provide a way for grouping subjects or targets.

Ponder specifies the following types of polices for expressing access control,

- **Authorization policies :** These are the access control policies specifying what targets a subject can access. The policy can express both positive and negative authorizations. The positive authorization policies specify what actions a subject can perform while negative authorization specifies those actions a subject is forbidden from performing. The Figure 1.1 gives the syntax of the authorization policy.

  The university policy can be represented in the Figure 1.2 as,

- **Information filtering policies :** These policies place restrictions on the actions performed. They can be used to provide an additional level of restriction in addition to an authorization policy that grants an action.

- **Delegation polices :** This policy enables one user to delegate access rights to another user.

- **Refrain policies :** Refrain policies define the actions that subjects must not perform on target objects even though they may actually be permitted to perform the action. They are similar to negative authorization policies but are enforced on the target rather than on the subject.

- **Obligation policies :** These policies specify the actions that need to be performed by managers when certain events occur within the system.

Ponder also supports various constraints like basic policy constrains and meta-policy constraint. Basic policy constraints are expressed in terms of a predicate which has to evaluate to true for the policy to apply. Meta-policies are used to specify policies about policy and the constraints are on self management and separation of duty. With all the above features, a large enterprise can structure its access control policy. Ponder also provides other features to enable the ease of management of large complex policies. We can specify groups for packaging related

policies, roles for semantic grouping of policies with common subjects. Also, they support policy hierarchies and the policy types can be specialized and reused. Relationships can also be defined showing the definition of roles participating in interactions.

The Platform for Privacy Preferences (P3P)

The Platform for Privacy Preferences (P3P) is a specification from the World Wide Web Consortium (W3C) for specifying the privacy policies of enterprises. Though the specification is platform independent and can be used across enterprises, it is not a general purpose specification. The P3P policies are higher level policies usually published by an enterprise to reveal their privacy practices to customers.

Enterprise Privacy Authorization Language (EPAL)

Enterprise Privacy Authorization Language (EPAL) was developed at International Business Machines (IBM). It is submitted for review to W3C. EPAL is mainly designed as a privacy policy interoperability language suitable for exchange between enterprises in a structured format. The language is appropriate for representing the data-handling practices and policies within and between enterprises that want to have a systematic way of managing privacy. This is also useful for automatic audit control of the accesses to the information and also for enforcing accountability of privacy practices. EPAL defines the attributes as a list of

```
<rule id=‘‘univ-policy’’ ruling=‘‘allow’’>
<user-category refid=‘‘faculty’’ />
<data-category refid=‘‘student-information’’ />
<purpose refid=‘‘view-and-assign-grades’’ />
<action refid=‘‘view, assign’’ />
<condition refid=‘‘condition’’ />
</rule>
```

Figure 1.3: EPAL policy example

hierarchies of,

- **data-categories :** This specifies the different ways in which the different data collected by an enterprise is used depending on the sensitivity of the

data. For example, the medical-record data is more sensitive than the contact information.

- **user categories :** This categorizes the different users of the data. In the above example, the medical record information is used by the doctor and the contact information is used by the sales department.

- **purposes :** This specifies the purpose for which the categorized data is used by the categorized user. The doctor will use the medical record for purpose of scheduling tests and the sales department will use the contact information for shipping purposes.

They also define actions, obligations and conditions. Actions specify how the data is used, obligations specify what must be satisfied in the environment and conditions must evaluate to true in the context for the rule to be applicable. An EPAL policy is a list of rules that are ordered according to descending precedence.

The Figure 1.3 gives the example for an EPAL policy

A study comparing XACML and EPAL concludes that EPAL uses a lot of XACML and that EPAL is a subset of XACML except for some specific features. For instance, EPAL and XACML share the same framework of a policy made up of a series of rules. A rule is applicable only if the condition in it evaluates to true and the effect of the rule is returned. Also, both languages share the same framework for the requests: a request is made up of a collection of attribute values.

1.2.4 Policy Testing Techniques

Martin et al [14] have developed a systematic method for testing access control policies which we have taken in thesis for analysis. Theirs is the first work on defining and measuring structural coverage of access control policies for testing. They have developed a coverage measurement tool for measuring policy coverage given a set of XACML policies and set of requests. Their coverage criterion is based on the structure of the policies and is similar to statement coverage in a program. The request generation process is random and the requests are got by

setting bits in a vector of policy attribute values. Even though the random request generation technique does not repeat requests that are already generated, this method has the disadvantage of using the random test input selection strategy. They use a tool to greedily reduce requests from the generated set of requests based on the coverage measure. They also perform mutation testing to analyze the fault detection capability of the reduced set of requests. Techniques [15] have been proposed to leverage mutation testing to automatically generation and/or reduce test sets for general purpose programming languages. L. J. Morell [16] gives the brief discussion in fault based testing for software, that is used frequently now a days. Software abstractions [17] book written by Daniel Jackson give idea about logic ans analysis In our approach, we developed automatically generation of requests based on Change-Impact analysis tool i.e. Margrave and Mutation testing technique for testing the policies.

Another area where access control policy testing is done is firewalls protecting network resources. Al-Shaer et al [18] propose automated testing of firewalls with respect to their internal implementation and security policies. They propose a novel firewall testing technique using policy-based segmentation of the traffic address space, which can intelligently adapt the test traffic generation to target potential erroneous regions in the firewall input space. Though this method is efficient, it is applicable only to firewall polices because they have made the testing dependent on the structure of the access control policy in a firewall. However, the idea of analyzing the logs of packets/request cannot be applied as such to any general purpose access control system.

1.2.5 Formal Policy Analysis

A complementary approach to access control policy testing is to convert the policy to a logical representation and use formal analysis techniques for verification and analysis. Hughes and Bultan translated XACML policies to their logical representation in the Alloy language and checked their properties using the Alloy Analyzer. Using their translator and the Alloy analyzer, it is possible to check a policy which is implemented as a combination of sub-polices correctly reproduces the properties

of the sub policies. This approach, though produces good results does not scale well with increase in the size of the policy. Zhang et al propose a mechanism for evaluating XACML polices through model checking. They evaluate whether the policies give legitimate users enough permissions to reach their goals and also to check whether the policies prevent intruders from reaching their malicious goals. However, the access control polices have to be translated to the RW language to apply their techniques. The limitations of these above approaches are that they do not treat all the features of XACML. Also, a predefined set of properties about the policy should be given which, does not exist in practice. Also, this analysis can become intractable when there are more attributes in the policy. The advantage of using testing is that no translation to a separate domain is needed to check the policies [19]. Also, all features of XACML can be tested.

Margrave is an efficient tool that enables checking for semantic consistencies in the policy and returns counter examples representing cases which are causing violation of properties of the policy. Change impact analysis is done between two policies to determine the properties of the policy. They construct a multi-terminal binary decision diagram to represent the rules in the policy. However this tool does not support all features of XACML.

We have implemented the framework [22] for testing access control policies by generating good quality of request suites and mutation testing method. Hennessy and Power [30] propose a strategy for the construction of test suites for grammar based software. The reduction criterion they use is based on the rule coverage of the test suites. They analyze if the code coverage and fault detection capability are reduced because of the reduced test suite. Martin et al [19] have developed a tool for the automated testing. They define a coverage measure for the different condition. Kapfhammer and Soffa [20] define a framework for testing database driven applications and the control flow between various entities in such an application. They define the test adequacy criteria for the database application based on the database interaction flow graph showing the interaction between the various entities.

1.3 Problem Definition

Sensitive data are increasingly available on-line through the Web and other distributed protocols.

To increase the confidence in the correctness of specified policies, policy developers can conduct policy testing by supplying typical test inputs (request) and subsequently checking test output (responses) against expected ones to enhance the correctness of specified policies [towards]. Testing of Access Control Policies along with the Application program is not a worthful practice. Unlike Software Testing we have the tools and technique for Access Control Policy Testing. Unfortunately, manual testing is tedious and time consuming job [6].

1.4 Objective

Our goal is to find out a significant framework by which we can assure the correctness of Access Control Policies for the better application development [19]. We need to generate better tests set (requests) for better testing result and comapre the results with the existing technique. In this thesis we focus on these areas.

1.5 Thesis Layout

The Thesis is organized as follows, **Chapter 2** provides the information on various components of an access control system and policy specification languages. In **Chapter 3**, we explain the framework for access control policy testing which includes request generation process and policy testing process. Implementation of the framework is explained in **Chapter 4**. **Chapter 5** presents the results of our evaluation and comparison with other method. We conclude and give direction for future work followed by bliographies.

Chapter 2

Access Control Polices and Enforcement

Introduction

Security Policy

Conclusion

Chapter 2

Access Control Polices and Enforcement

2.1 Introduction

In this chapter, we give an overview of the various components that make up an access control system and an introduction to policy specification languages [6]. The Figure 2.1 shows the various functional components of any system protecting its resources by enforcing access control. The user makes a request to the entity protecting the resources in the system, the Policy Enforcement Point (PEP). The PEP forms the appropriate access control request in a format applicable to the Policy based on the attributes of the requester, the action sought, the resource requested and the environment and gives it to the Policy Decision Point (PDP). The PDP looks up the policy that applies to the request and returns a response to the PEP. The PEP then returns the corresponding decision to the requester. The advantage of using this abstract model is that any application can use this system.

There can be various vulnerabilities in a system implementing access control. For example, the user has to first be properly authenticated into the system. Then the PEP should correctly perform the translation from the user or application specific request to that specific to the policy. This is vulnerability because the policy specification language may be more expressive for specifying an application's security requirements. For example, XACML allows a set of subjects to request access to a set of resources. But an application can have a strict requirement that

only one subject can access one resource at a time. In these cases, the PEP implementation should be correctly implemented to be aware of this restriction when performing the translation from the user's request to a policy specific request. Next, the access control policies have to correctly specify the intended behaviour of the system. Also, the PDP has to perform the evaluation correctly.

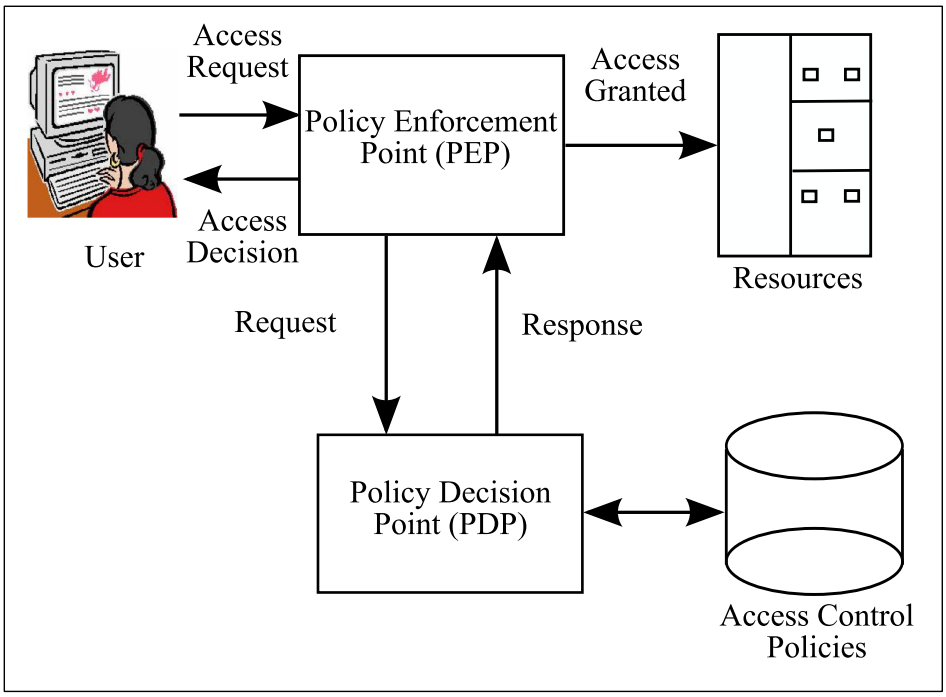

Figure 2.1: Functional Components of an Access Control System

Among these vulnerabilities, one of the most basic requirements is to ensure that the security policy is specified correctly.

In this thesis, we focus on the problem of ensuring that the access control policies are specified correctly. A policy is considered to be correctly specified when it satisfies all the properties of the system. An example of a property is that a particular subject should not access certain resources. These properties can be explicitly and formally expressed and formal analysis techniques like resolution theorem proving can be used to prove if a property holds in a policy. However, such properties of a policy do not exist in practice and it is difficult to infer such properties in a large system. Also, the formal analysis techniques are not scalable. A practical way for ensuring the correctness of the policy is to test the policy

against a set of requests and check if the responses obtained are as expected. This is the policy testing approach which is followed in this work [11].

2.2 Secutiry Policy

The security policy in an access control system provides a systematic way for specifying the strategy and practices for ensuring the security, integrity and availability of resources in an information system. In this section, we will give a brief overview of policy specification languages and describe XACML which we will be using for illustrating our approach to testing [6].

2.2.1 Policy Specification Languages

Previously access control policies were written by hard coding directly into the program by the programmer. Later on, as the policies became more and more complicated, separate policy specification languages were developed. There are many policy specification languages and they can be either generic or specific to applications. Generic policy specification languages are designed for enforcing access control in broad domains like distributed policy management, protecting the privacy of enterprises, etc. Jajodia et al [21] propose a logical language for a model that allows the specification of different access control policies.In [22] they have used XACML as language and tool for testing, an access control language for web services [23], etc.

The framework for policy testing is general and can be used for testing access control policies specified in other rule-based systems. In this thesis we present the approach in the context of one of the generic specification language XACML.

2.2.2 XACML

XACML provides a standardized way of expressing authorization polices and a standard format for expressing queries over these policies [19].

We have chosen to illustrate our approach to policy testing using XACML because it is a general purpose specification language with various advantages for

enforcing access control like,

- It is an open source standard ratified by the Organization for the Advancement of Structured Information Standards (OASIS). Because it is a standard, the various features of XACML have been examined by experts and so the specification is stable. Also, it is expected to be used widely in the industry because of its ratification.

- XACML is specified in XML format which is used for e-business applications for Electronic Data Interchange (EDI) in business-to-business and business-to-consumer transactions. Because of this, these applications can be easily configured to exchange or share XACML policies for enforcing access control.

- The specification is flexible and extensible. It is flexible since it is a generic standard and can be applied for specifying policies in all applications. It is extensible because the data types, functions, attribute types, and the way for combining multiple applicable rules/polices can be extended. Also currently there is work on developing an XACML profile for Web Services, SAML and LDAP. This shows the language is adaptable to different environments.

- It is a portable standard. Since the specifications are in XML format, it can be used across applications.

- Conceptually it follows the PDP and PEP model which makes it applicable to many application environments.

- It supports distributed policies. The security policy of an enterprise may be enforced at different points and there is a need for specifying distributed policies.

XACML follows the abstract model as shown in Figure 2.1 for policy enforcement defined by the Internet Engineering Task Force (IETF) [24].

The specification defines the PEP (Policy Evaluation Point) and PDP (Policy Decision Point) as any other access control implementation. The request is given to a PEP which processes it and converts it to an XACML request format and

gives it to the PDP. The PDP has access to the policies and it gets the request and determines if it has to give access to the policy or not. The PDP and PEP implementation is dependent on the application. They may be in the same application or be as separate entities on different applications or be available as a service over a network.

XACML Constructs

All XACML polices contain either a Policy or Policy Set as the basic element. A Policy is composed of a set of Rules. A set of policies or policy sets are combined to form a Policy Set. Figure 2.2 shows these main components and the hierarchical relation between Policy Sets, Policies, Rules and Conditions in XACML. When there are multiple rules in the policy and multiple policies/policy sets in a policy set, it is possible that a single access request can be applied to multiple rules to return conflicting access decisions. The way these conflicts must be resolved is dependent on the specific application's policy. However, XACML specifies some standard rule and policy combining algorithms for this. They are,

- **First Applicable :** Among the set of rules (policies), this returns the effect of the rule (policy) that first evaluated to true. Here, the ordering of the rules is important.

- **Permit Overrides and Deny Overrides :** In the set of rules (policies) in a policy (policy set), if a rule (policy) evaluates to true and if its effect is permit then the result of the rule (policy) combination is permit. If the effect of the rule (policy) is deny or if it is not applicable then all the rules (policies) in the set are evaluated to check if there is any permit rule (policy) evaluating to true. If such a rule exists, a permit decision is returned, if not a deny decisions is returned if none of the permit rules are applicable. Hence, here permit rules are given precedence. Similarly the deny overrides rule (policy) combining algorithm is defined.

- **Only One Applicable :** This combining algorithm is defined for combining policies in a policy set. If no policy is applicable or more than one policy is applicable, the result is defined as not applicable and in determinant. If

only one policy in the policy set is applicable for a request then the result of the policy is returned.

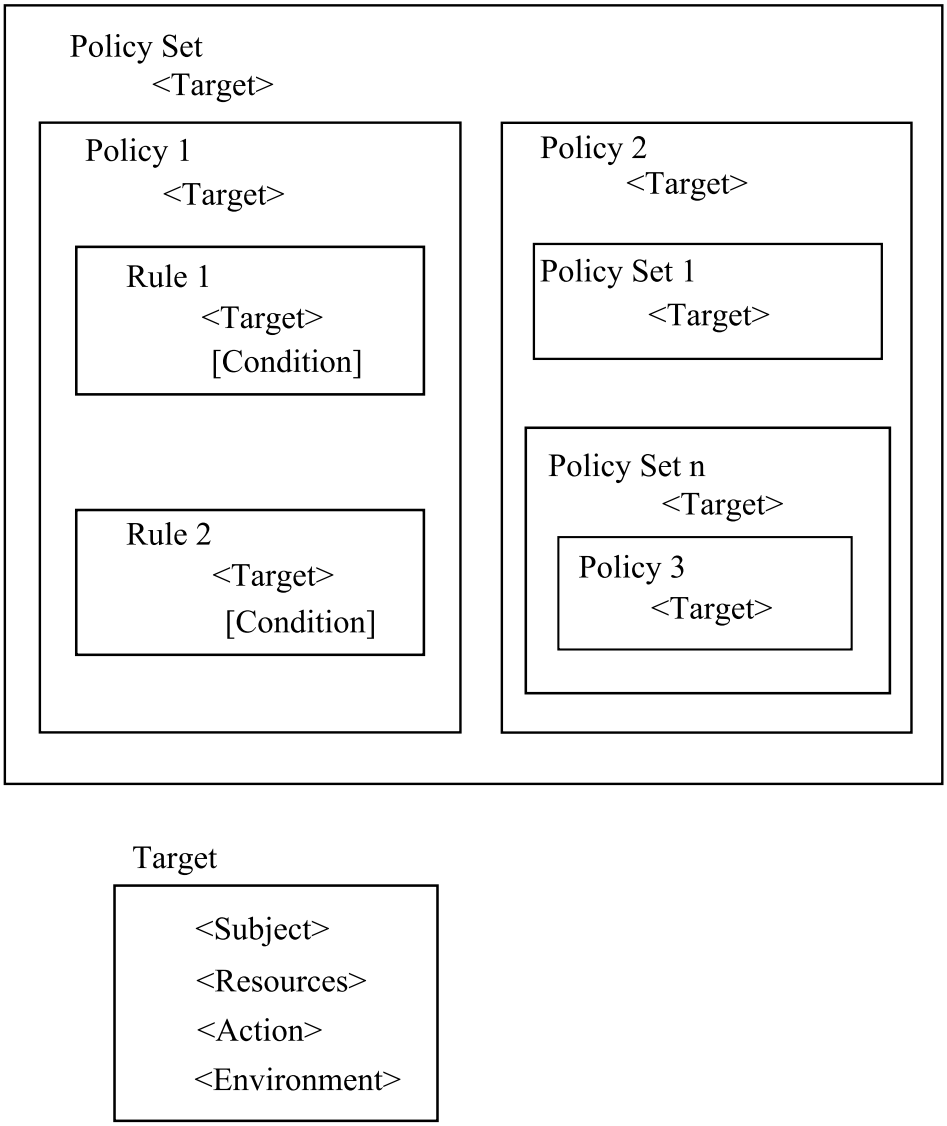

Figure 2.2: Components of XACML Policy

The permit overrides and deny overrides combining algorithms can be specified to be ordered requiring that the rules be evaluated in the order in which they are specified. In addition to the above, user-defined combining algorithms can also be added. A Rule is the most elementary unit of a Policy. A rule is made of the elements, Target, Effect and the optional Condition element. A Target defines a set of Subject, Resource and Actions elements for which the rule is intended to apply. The Effect specifies the access decision as permit or deny that is returned on

the successful evaluation of the rule. The Condition element can include complex functions that further refine the applicability of the rule. The policy itself can have a target specifying the applicability of the policy. In this case, this target can be thought of as an index into the policy. The index can be the common criteria that have to be satisfied for the set of rules in the policy to be applicable. When there are a number of policies each with a number of rules, the index of each policy helps to speed up the evaluation of a decision request by first checking the applicable policy targets and then evaluating the rules in those policies. A policy set may be used for semantically grouping policies/policy sets. For example, grouping those policies defining authorization for a particular object/subject, etc. A policy set also includes a target element which again be used as an index for checking the applicability before evaluation of all the constituent elements [11].

The other essential constructs in an XACML policy are attributes, attribute values and functions. Attributes are named values of known types. The Subject, Resource, Action and Environment of a given access requests are described by attributes. A request will mostly contain a set of attributes. These are compared with the corresponding attribute values in the policy and an access decision is made. A request can match the attributes in the policy by using the AttributeDesignator type which identifies attributes by their name and type. The AttributeSelector is used for matching a request with the attribute values in the policy through an XPath query. The attributes values can be operated on by a number of functions like string comparison, date and time functions, logical functions, numeric conversions, set functions, bag functions, etc and the values returned can be compared to arrive at an access decision [6].

An Example Policy

In this section, we describe an example XACML policy [14] in a university. Figure 2.3 shows a simplified form of this policy with the major XACML components. This policy describes the way in which the access to the grade resource is controlled in a university. The policy has an empty target, which means that by default the set of rules in the policy are applicable for any request. The target of

```
01<Policy PolicyId="univ" RuleCombinationAlgId="permit-overrides">
02 <Target>
03  <Subjects> <AnySubjects/> </Subjects>
04  <Resources><Resource> <AnyResource/> </Resource></Resources>
05  <Actions> <AnyAction/> </Actions>
06 </Target>
07 <Rule RuleId="1" Effect="Permit">
08 <Target>
09  <Subjects><Subject> Faculty </Subject></Subjects>
10  <Resources> Grades </Resources>
11  <Actions><Action> Assign </Action>
12  <Action> View </Action></Actions>
13 </Target></Rule>
14 <Rule RuleId="2" Effect="Deny">
15 <Target>
16  <Subjects><Subject> Student </Subject></Subjects>
17  <Resources>Grades </Resources>
18  <Actions><Action> Assign </Action></Actions>
19 </Target>
20 </Rule>
21 <Rule RuleId="3" Effect="Permit">
22 <Target>
23  <Subjects><Subject> Student </Subject></Subjects>
24  <Resources> Grades </Resources>
25  <Actions><Action> View </Action></Actions>
26 </Target>
27 </Rule>
28 <!-- A final, "fall-through" rule that always Denies -->
29 <Rule RuleId="FinalRule" Effect="Deny"/>
30</policy>
```

Figure 2.3: An example XACML policy

the policy in general is used as an index for the rules. In the above example, the target could have been used to restrict the policy to be specific to a university as shown in Figure 2.4.

There are three rules in the policy and a final fall through rule. The three rules are combined based on the permit overrides rule combining algorithm. This means that set of rules are combined giving precedence to rules with an effect of permit.

```
<Target>
<Subjects> <Subject> NC State University User </Subject> </Subjects>
<Resources> <Resource> NC State University Academic Records
</Resource> </Resources>
<Actions> <AnyAction/> </Actions>
</Target>
```

Figure 2.4: XACML policy target for a university's policy

```
<Request>
<Subjects> Student </Subjects>
<Resources><Resource>Grades</Resource></Resources>
<Actions> View </Actions>
</Request>
```

Figure 2.5: XACML Request

```
<Response>
<Result>
<Decision>Permit</Decision>
</Result>
</Response>
```

Figure 2.6: XACML Response

XACML Request and Response context

The XACML Request and Response context specifies the standard format with which a request and response from the system is got. The Figures 2.5 and 2.6 show a simplified form of an XACML Request and Response. The request enables a set of subjects, resource and action elements to be specified. In this example, the student requests access to view the grade resource. The response is returned on evaluating the request against the set of rules in the policy. In this example, a decision is returned on matching the attribute values in the request with the attribute values in the rule 3 in the policy.

2.2.3 MARGRAVE tool

In this paper, we used a suite called Margrave for Change-Impact Analysis of Access Control Policies written in the XACML standard. Margrave has two components [6]:

1. A verification system that consumes a policy and property and determines whether the policy satisfies the property. (More generally, this can be used as a query engine to investigate the behavior of a policy.)

2. A system for change-impact analysis. The analysis consumes two policies that span a set of changes and summarizes the differences between the two

policies. Users can not only examine the summary, but also query it and verify properties of the change. This verification can happen even in the absence of formal properties about the system as a whole. (Indeed, these properties may not even hold of the entire system.)

We have implemented the idea of change-Impact analysis for generating the request suites. Let's see how the change-Impact analysis component works.

Change-Impact Analysis

Given two versions of a policy, change-impact analysis tool outputs counterexamples that illustrate semantic differences between the two policies. More specifically, each counterexample represents a request that evaluates to a different response when applied to the two policy versions. For example, a particular request revaluates to permit for policy p but the same request evaluates to deny for policy p'. Change-impact analysis is usually performed on mature policies that are undergoing maintenance or updates to avoid accidental injection of anomalies. In our case, we exploit the functionality of change-impact analysis to automatically generate access requests by iteratively manipulating the inputs to a change-impact analysis tool. We use tool Margrave's API [25] to perform a change-impact analysis on the original policy and each of the policy versions. Based on the counterexample produced by Margrave, the request generator generates request. Exactly one request is generated from each version. Margrave package running in PLT scheme with Drscheme package and for generation counterexample CUDD tool [26] is necessary.

2.3 Conclusion

In this chapter we studied the basic languages tool and concepts which is required for testing the Access Control Policies. We have seen basic Functional Components of an Access Control System, Security Policies, XACML (eXtensible Access Control Markup Language) in which we made access control policies, and MARGRAVE tool which is used to perform change-impact analysis for request generation.

Chapter 3

Policy Testing Framework

Chapter 3

Policy Testing Framework

3.1 Introduction

In this chapter, we describe the framework [22] followed for policy testing. We also describe the technique for policy testing in brief. We have given name Access Control Policy Checker (ACPC) to that testing framework. ACPC has used the concept of mutation testing, generally used in software testing and Change-Impact Analysis method for generating the good quality of request suites.

3.2 Proposed Policy Testing Framework

ACPC (Access Control Policy Checker) is the model [14] for the automated testing the correctness of the Access Control Policies. This model will work for the policies written in XACML and having two sections.

1. In first section we generate the sets of Requests
2. In second section we check the correctness of the Policies.

Figure 3.1 shows the testing framework called ACPC (Access Control Policy Checker) for testing the correctness of policy. The input to the framework is the access control policy that is to be tested. In the request generation phase, this policy is converted into derived policies and performing the Change-Impact analysis. The output of this phase is the request suites. In the policy checker phase, the input is request suites, generated by request generation phase and the policy

for testing. In the policy checker phase mutation operator is used for producing mutant (faulty) policy and after generating the response against the request we compare and evaluate the result for mutant killing, the higher the mutant killing rate the higher the correctness of the policy under test, so we can say:

$$Mutant\_Killing\_Rate\ \alpha\ Correctness\_of\_the\_Policy$$

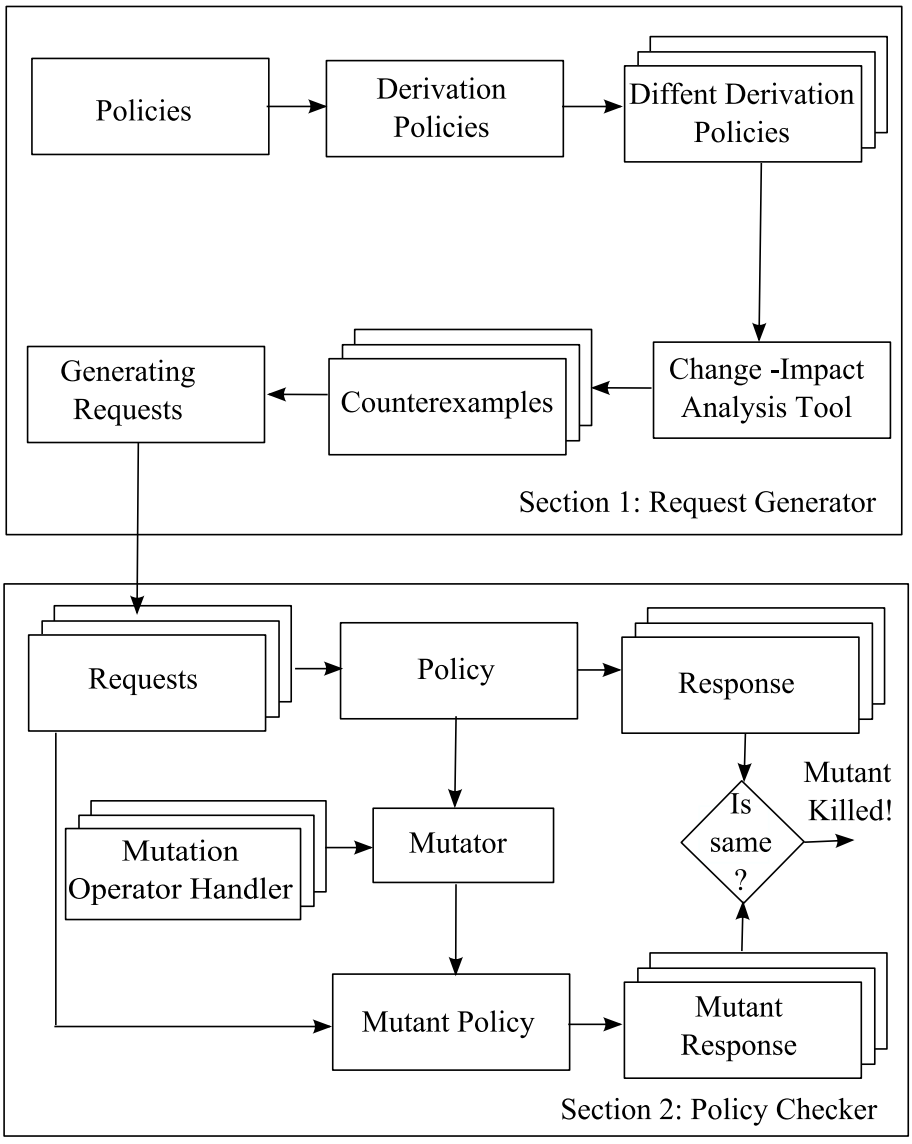

Figure 3.1: ACPC (Access Control Policy Checker) Model

3.3 Request Generation Process

To automatically generate high-quality request suites for access control policies, we implemented framework based on change-impact analysis. Figure 3.2 shows the overview of the framework. The framework receives a set of policies under test and outputs a set of request for policy authors to inspect for correctness. The framework consists of three major components: derivation, change-impact analysis and request generation. The key notion of the framework is to derived two versions of the policy under test in such a way that test coverage targets (e.g., certain policies, rules, or conditions) are encoded as the differences of the two derived versions. A change-impact analysis tool can then be leveraged to generate counterexamples to witness these differences. Based on the generated counterexamples, the framework generates requests [14].

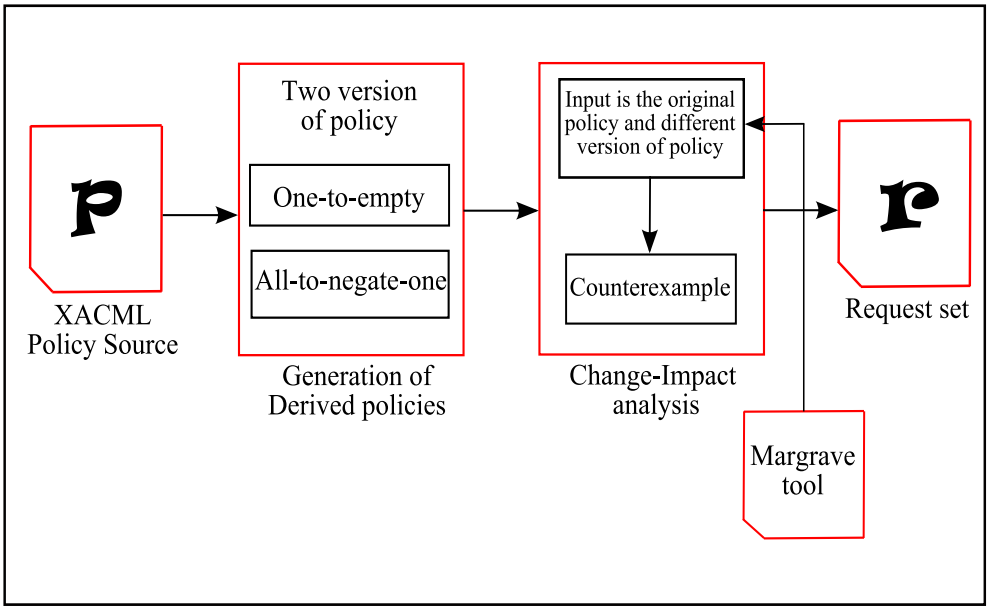

Figure 3.2: Request Generation Process Framework

3.3.1 Derivation

Given the policy under test, the derivation component derives the policy's versions, which are later fed to a change-impact analysis tool. The goal is to formulate the inputs to the change-impact analysis tool so that specifically targeted parts

of the policy under test are covered. We provide two variants of version below called one-to-empty and all-to-negate-one. We discuss their analysis cost and the situations where they may not work well. Although the framework has been developed to support multiple policies, to simplify illustration we describe the derivation variants with the case of a single policy p that contains n rules. To further illustrate the framework, we provide a concrete example XACML policy in Figure 3.3. An XACML policy encodes rules in XML syntax. Each rule has a set of constraints found in the Target elements that must be satisfied by a request in order for that rule to be applied. This example policy has two rules: the first one denies access requests for "dissemination" of the "demo:5" resource and the second one permits all other access requests. The first rule is defined by the Rule element on Line 2 and the Target element on Lines 3-21. The second rule is defined by the Rule element on Line 23. When multiple rules can be applied on a request, the decision of the first applicable rule will be returned (as specified by the "first-applicable" rule combining algorithm on Line 1) [14].

one-to-empty : For each rule r in p, the two synthesized versions are an empty policy and a policy that contains only r. If r is a permitting rule, the synthesized empty policy is an empty denying policy. If r is a denying rule, the synthesized empty policy is an empty permitting policy. The reason for this mechanism is as follows. Comparing a permitting rule r with an empty permitting policy will not help generate requests to cover r because no counterexamples are generated for these two versions. Similarly, comparing a denying rule r with an empty denying policy will not help generate requests to cover r. This synthesis process is applied n times. So there are n pairs of policy versions synthesized for p. Consider the example policy written in XACML in Figure 3.3. The first pair of policy versions synthesized for this policy is an empty permitting policy and the original policy with Line 23 removed (i.e., the remaining rules). Applying change-impact analysis on each pair has low cost because each version contains only a single rule. Note that this variant does not take into account the interactions among different rules unlike the all-to-negate-one variant below.

```
1<Policy Id="demo" RuleCombAlgId="first-applicable">
2 <Rule RuleId="1" Effect="Deny">
3 <Target>
4   <Subjects> <AnySubjects /> </Subjects>
5   <Resources>
6       <Resource>
7           <ResourceMatch MatchId="equal">
8           <AttrValue>demo:5</AttrValue>
9           <ResourceAttrDesignator AttrId="objectid" />
10          </ResourceMatch>
11      </Resource>
12  </Resources>
13  <Actions>
14      <Action>
15          <ActionMatch MatchId="equal">
16          <AttrValue>dissemination</AttrValue>
17          <ActionAttrDesignator AttrId="actionid" />
18          </ActionMatch>
19      </Action>
20  </Actions>
21 </Target>
22 </Rule>
23 <Rule RuleId="2" Effect="Permit" />
24</Policy>
```

Figure 3.3: An example XACML policy

all-to-negate-one : For each rule r in p, the two synthesized versions are p and p where the decision of r is negated. This process is applied n times so there are n pairs of policy versions synthesized for p. Again, consider the example policy in Figure 3.3. The first pair of policy versions synthesized for this policy is the original policy and the original policy with the effect on Line 2 changed to "Permit". Applying change-impact analysis on each pair has higher cost than the one-to-empty variant because the analysis complexity is heavily dependent on the size of the two versions rather than the differences between the two versions. Note that this variant takes into account interactions among different rules. This variant should be at least as good as the one-to-empty variant in terms of achieving policy structural coverage and fault detection but it will have a higher computational

cost, especially for large, complex policies.

The preceding two variants are specifically developed for achieving high rule coverage. Because the coverage of a rule implies the coverage of the policy that contains the rule, our two variants also indirectly target at achieving high policy coverage [22].

3.3.2 Change-Impact Analysis

Given two versions of a policy, Change-Impact analysis tool [14] outputs counterexamples that illustrate semantic differences between the two policies. More specifically, each counterexample represents a request that evaluates to a different response when applied to the two policy versions. For example, a particular request r evaluates to permit for policy p but the same request evaluates to deny for policy p'. Change-impact analysis is usually performed on mature policies that are undergoing maintenance or updates to avoid accidental injection of anomalies. In our case, we exploit the functionality of change-impact analysis to automatically generate access requests by iteratively manipulating the inputs to a change-impact analysis tool.

We use tool Margrave's API to perform a change-impact analysis on the original policy and each of the policy versions. Based on the counterexample produced by Margrave, the request generator generates request. Exactly one request is generated from each version. Margrave package running in PLT scheme with drscheme package and for generation counterexample CUDD tool is necessary.

3.3.3 Request Generation

Given two policies, a change-impact analysis tool outputs counterexamples that are evaluated to different responses against these two policies. We generate requests based on these counterexamples. Some change-impact analysis tools may produce abstract counterexamples, which are not immediately ready to be translated into a concrete request. For example, a change-impact analysis tool may produce an abstract counterexample [27] for the policy in Figure 3.3 like if ((resource == demo:5) (action== dissemination)), deny becomes permit. Then we

need to solve the constraint and derive one request (or optionally more requests) for the constraint. Other change-impact analysis tools may produce counterexamples at the concrete level, being the same as the level of requests. Our implementation leverages a change-impact analysis tool that produces counterexamples at the concrete level so we do not need to refer to a constraint solver to map from counterexamples to requests.

3.4 Policy Testing Process

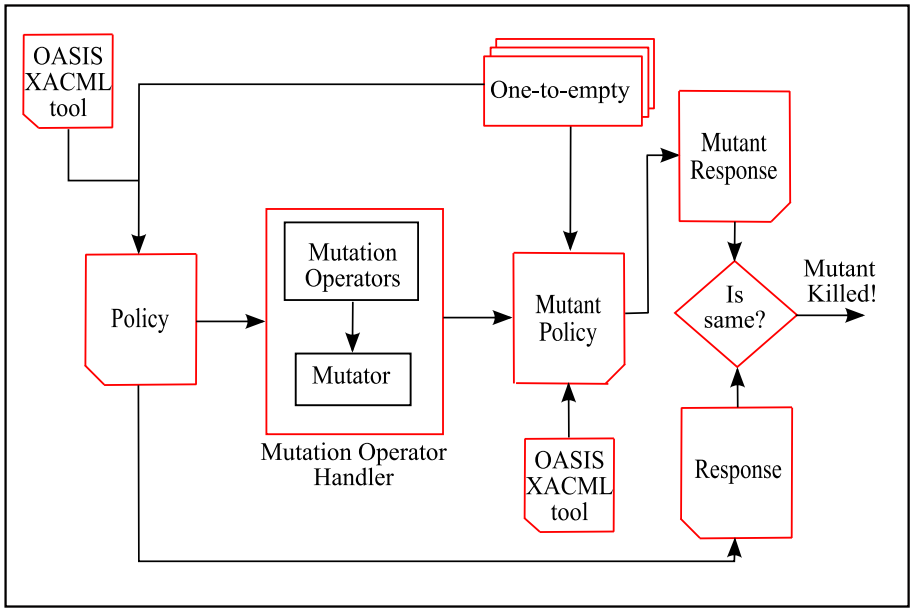

Figure 3.4: Policy Mutation Testing Framework

Mutation testing has historically been applied to general purpose programming languages. The program under test is iteratively mutated to produce numerous mutants, each containing one fault.

A test input is independently executed on the original program and each mutant program. If the output of a test executed on a mutant differs from the output of the same test executed on the original program, then the fault is detected and the mutant is said to be killed.

The fundamental premise of mutation testing as stated by Geist et al. [28] is that, in practice, if the software contains a fault, there will usually be a set of

mutants that can only be killed by a test that also detects that fault. In other words, the ability to detect small, minor faults such as mutants implies the ability to detect complex faults. Because fault detection is the central focus of any testing process, mutation testing provides an external measure of the effectiveness of that process. The higher the percentage of killed mutants, the more effective the test set is at fault detection.

In policy mutation testing, the program under test, test inputs, and test outputs correspond to the policy, requests, and responses, respectively. An overview of our framework for policy mutation testing is illustrated in Figure 3.4. In the framework, we first define a set of mutation operators, whose details are described in Section 3.4.1. Given a policy and a set of mutation operators, a mutator generates a number of mutant policies. Given a request set, we evaluate each request in the request set on both the original policy and a mutant policy. The request evaluation produces two responses for the request based on the original policy and the mutant policy, respectively. If these two responses are different, then we determine that the mutant policy is killed by the request; otherwise, the mutant policy is not killed.

Unfortunately, there are various expenses and barriers associated with mutation testing. The first and foremost is the generation and execution of a large number of mutants. For general-purpose programming languages, the number of mutants is proportional to the product of the number of data references and the number of data objects in the program [29]. For XACML policies, the number of mutants is proportional to the number of policy elements, namely policy sets, policies, targets, rules, conditions, and their associated attributes.

3.4.1 Mutation operator Handler

Mutation operators describe modification rules for modifying access control policies to introduce faults into the policies. Previous studies [30] have been conducted to investigate the types and effectiveness of various mutation operators for general-purpose programming languages; however, these mutation operators often do not directly apply to mutating policies. This section describes the chosen mutation

Table 3.1: Index of Mutation Operators

ID	Description
PSTT	Policy Set Target True. The policy set is applied to all request.
PSTF	Policy Set Target False. The policy set is not applied to any requests.
PTT	Policy Target True. The policy is applied to all requests.
PTF	Policy Target False. The policy is not applied to any requests.
RTT	Rule Target True. The rule is applied to all requests.
RTF	Rule Target False. The rule is not applied to any requests.
RCT	Rule Condition True. The condition always evaluates to true.
RCF	Rule Condition True. The condition always evaluated to false.
CRE	Change Rule Effect. The rule effect is inverted (e.g. permit for deny).

operators for XACML policies that implement our fault model. An index of the mutation operators is listed in Table 3.1 and their details are described below. The first eight mutation operators emulate syntactic faults because these mutation operators manipulate the predicates found in the target and condition elements. In particular, PSTT, PSTF, PTT, PTF, RTT, RTF, RCT, and RCF emulate syntactic faults as simple typos in the policy set, policy, and rule target elements as well as the condition elements which result in the predicates found in those elements to always evaluate to true or false [14].

The last mutation operator CRE, emulate semantic faults because they manipulate the logical constructs of XACML policies.

Policy Set Target True (PSTT) : Ensure that the policy set is applied to all requests by removing the `<Target>` tag of each PolicySet element. The number of mutants created by this operator is equal to the number of PolicySet elements with a Target tag.

Policy Set Target False (PSTF) : Ensure that the policy set is never applied to a request by modifying the `<Target>` tag of each PolicySet element. The number of mutants created by this operator is equal to the number of PolicySet elements.

Policy Target True (PTT) : Ensure that the policy is applied to all requests simply by removing the `<Target>` tag of each Policy element. The number of

mutants created by this operator is equal to the number of Policy elements with a Target tag. Policy Target False (PTF). Ensure that the policy is never applied to a request by modifying the `<Target>` tag of each Policy element. The number of mutants created by this operator is equal to the number of Policy elements.

Rule Target True (RTT) : Ensure that the rule is applied to all requests simply by removing the `<Target>` tag of each Rule element. The number of mutants created by this operator is equal to the number of Rule elements with a `<Target>` tag.

Rule Target False (RTF) : Ensure that the rule is never applied to a request by modifying the `<Target>` tag of each Rule element. The number of mutants created by this operator is equal to the number of Rule elements.

Rule Condition True (RCT) : Ensure that the condition always evaluates to True simply by removing the condition of each Rule element. The number of mutants created by this operator is equal to the number of Rule elements with a `<Condition>` tag.

Rule Condition False (RCF) : Ensure that the condition always evaluates to False by manipulating the condition value or the condition function. The number of mutants created by this operator is equal to the number of Rule elements.

Change Rule Effect (CRE) : Invert each rule's Effect by changing Permit to Deny or Deny to Permit. The number of mutants created by this operator is equal to the number of rules in the policy. This operator should never create equivalent mutants unless a rule is unreachable, a strong indication of an error in the policy specification.

These entire mutation operators are used for our Mutation testing framework. The Mutation operator handler contain one more component called Mutant, which is responsible for creating the mutant (faulty) policy, by taking input as the original policy and mutation operator.

3.4.2 Mutant and Original Policy Verification

In this section, we will discuss the verification of original XACML policy and Mutant XACML policy. After getting the mutant policy generated by Mutation

Operator Handler, we need the response against all the requests generated by Request Generation phase. Therefore, we need the OASIS XACML tool for verification and getting the response. The working principle of OASIS XACML tool is; It gives the response when we provide policy and request as input to the tool the output of the OASIS XACML tool is the response in XACML language as we have seen in **Chapter 2**. We use this principle for verifying the Mutant policy and Original policy by supplying the same request set for getting the response. This is very simple process and it worked just like Access Control System.

3.4.3 Difference Checker

The work of difference checker is very simple; it only compares the response of the Mutant policies and Original policy against same request set. It returns 1 if both the response is different otherwise return 0. The mutant killing rate is calculated by number of 1, if the number of 1 is more we say that the mutant killing rate is more. We can formulate the mutant killing rate as;

$$Mutant\_Killing\_rate(mk\_rate) = \frac{\#Mutant\_Killed}{\#Mutant\_Policies}$$

The *mk_rate* is easily calculated with the help of above formula and by multiplying by 100 we can find the mutant killing percentage (i.e. mk%).

3.5 Conclusion

In this chapter, we have seen the detailed description about the ACPC (Access Control Policy Checker), which is the model for policy testing. ACPC have mainly two phases; first one is for generating the request set and the second phase is for performing the testing. In this chapter we have describe the nine mutation operators that we have used for performing the mutation testing.

Chapter 4

Implementation of ACPC

Introduction

Sample Policies

Request Sets

Mutation Testing

Conclusion

Chapter 4

Implementation of ACPC

4.1 Introduction

In this chapter, we describe the working principle and implementation detail of ACPC (Access Control Policy Checker). We have seen the working of different components of ACPC in the previous chapter. Now we will discuss about:

1. Different sample policies those have used as a standard for testing.
2. How request suites has been generated by first phase and,
3. How we test the policy in second phase, with the help of one XACML sample policy [22].

4.2 Sample Policies

We used nine XACML policies collected from three different sources as subjects in our experiment [14]. Table 4.1 summarizes the basic statistics of each policy. The first column shows the subject names.

Columns 2-5 show the numbers of policy sets, policies, rules, and conditions, respectively. Four of the policies, namely codeA, codeB, codeC, and codeD are examples used by Fisler et al. The remaining policies are examples of real XACML policies used by Fedora. Fedora is open source software that gives organizations a flexible service-oriented architecture for managing and delivering digital content. Fedora uses XACML to provide fine-grained access control to the digital content that it manages. The Fedora repository of default and example XACML policies

Table 4.1: List of Policies used in the Experiment

Policy	**#policy set**	**#policy**	**#rule**	**#condition**
codeA	5	2	2	0
codeB	11	5	5	0
codeC	8	4	4	0
codeD	11	5	5	0
default-2	1	13	13	12
demo-11	0	1	3	4
demo-26	0	1	2	2
demo-5	0	1	3	4
mod-fedora	1	12	12	10

provides a useful resource of realistic subjects. All these policies are suitable for implementing the mutation operator that we have chosen for mutation testing.

Figure 4.1 shows an example XACML policy, which is revised and simplified from a sample Fedora policy. This policy has one policy element, which in turn contains two rules. The rule combining algorithm is "first-applicable", meaning that the decision of the first applicable rule encountered during evaluation is returned. Lines 2-13 define the policy's target, which indicates that this policy only applies to those access requests of a resource "demo:5". The target of Rule 1 (Lines 15-25) further narrows the scope of applicable requests to those requesting to perform a "Dissemination" action on resource "demo:5". Its condition (Lines 26-35) indicates that if the subject's "loginId" is "testuser1", "testuser2", or "fedoraAdmin", then the request should be denied. Otherwise, according to Rule 2 (Line 37) and the rule combining algorithm of the policy (Line 1), a request applicable to the policy should be permitted. In the original demo-5.xml XACML policy, 3 rules are there, here we have taken two for the sake of creating example.

This example is describes here for the sake of readers to understand about the structure of the XACML policies. Next section is all about the actual implementation of our model with the help of Margrave tool and XACML tool.

```
01 <Policy PolicyId="demo" RuleCombinationAlgId="first-applicable">
02 <Target>
03 <Subjects> <AnySubjects/> </Subjects>
04 <Resources>
05 <Resource>
06 <ResourceMatch MatchId="equal">
07 <AttributeValue>demo:5</AttributeValue>
08 <ResourceAttributeDesignator AttributeId="objectid"/>
09 </ResourceMatch>
10 </Resource>
11 </Resources>
12 <Actions> <AnyAction/></Actions>
13 </Target>
14 <Rule RuleId="1" Effect="Deny">
15 <Target> <Subjects><AnySubject/></Subjects>
16 <Resources> <AnyResource/> </Resources>
17 <Actions>
18 <Action>
19 <ActionMatch MatchId="equal">
20 <AttributeValue>Dissemination</AttributeValue>
21 <ActionAttributeDesignator AttributeId="actionid"/>
22 </ActionMatch>
23 </Action>
24 </Actions>
25 </Target>
26 <Condition FunctionId="not">
27 <Apply FunctionId="at-least-one-member-of">
28 <SubjectAttributeDesignator AttributeId="loginid"/>
29 <Apply FunctionId="string-bag">
30 <AttributeValue>testuser1</AttributeValue>
31 <AttributeValue>testuser2</AttributeValue>
32 <AttributeValue>fedoraAdmin</AttributeValue>
33 </Apply>
34 </Apply>
```

Figure 4.1: An example of XACML Policy (demo-5.xml)

4.3 Request Sets

The request generation process is very important and very necessary process of any testing weather it's a software testing or is a access control policy testing. To show the process of request generation here we have taken an example so that the reader should understand the ACPC framework easily and can relate it. The sample XACML policy named RPSlist.xml shown in Appendix-A, is the simple policy to understand. In the RPSlist.xml policy there is sub policies named RPS_Faculty.xml, RPS_Student.xml, PPS_Faculty.xml, and PPS_Student.xml.

For generating request; first perform the Change-Impact analysis by Margrave tool, under which Drscheme is there which is the tool responsible for generating the counterexample. This counterexample is the raw request material in the form

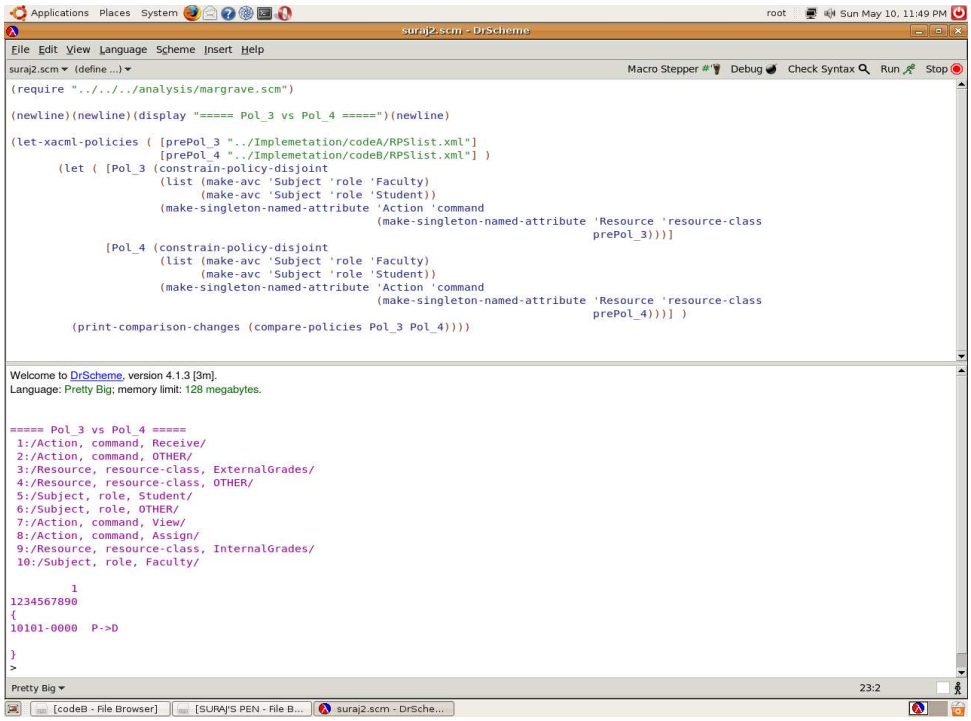

Figure 4.2: Generation of counterexample by Margrave tool named Drscheme

of code, again this code is processed and we generate the request.

Figure shows the counterexample generated by change-impact analysis tool, Margrave's Drscheme. In this if you will see the original policy RPSlist.xml is used and we have used the derivation function *all-to-negate-one* by which PPS_Student.xml *Permit* rule we have changed to *Deny*. Because of this derived policy we get P→D symbol in the counterexample shown in the Figure . The counterexample shown in the Figure is 10101-000 P→D, the code given in the form of 1 and 0. 1 indicates the presence of *action*, *resource* and *subject* to the counterexample. In this case *action* is "Receive", *resource* is "ExternalGrades" and *subject* is "Student", so our request in the form of XACML is shown in Figure 4.3.

This is the correct XACML format of request with three attributes subject, resource and action. This request we pass in to the second phase i.e. policy checker where we perform mutation testing.

```
<?xml version="1.0" encoding="UTF-8"?>
<Request xmlns='urn:oasis:names:tc:xacml:1.0:context' xmlns:xsi='http://
www.w3.org/2001/XMLSchema-instance'
xsi:schemaLocation='urn:oasis:names:tc:xacml:1.0:context
      cs-xacml-schema-context-01.xsd'>
 <Subject>
  <Attribute AttributeId='urn:oasis:names:tc:xacml:1.0:subject:subject-id'
DataType='http://www.w3.org/2001/XMLSchema#string'>
    <AttributeValue> Student </AttributeValue>
  </Attribute>
 </Subject>
 <Resource>
  <Attribute AttributeId='urn:oasis:names:tc:xacml:1.0:resource:resource-id'
DataType='http://www.w3.org/2001/XMLSchema#anyURI'>
    <AttributeValue> ExternalGrades </AttributeValue>
  </Attribute>
 </Resource>
 <Action>
  <Attribute AttributeId='urn:oasis:names:tc:xacml:1.0:action:action-id'
DataType='http://www.w3.org/2001/XMLSchema#string'>
    <AttributeValue> Receive </AttributeValue>
  </Attribute>
 </Action>
</Request>
```

Figure 4.3: Request generated in XACML form

4.4 Mutation Testing

Mutation testing is performed with the help of mutation operator that we have already seen in the previous chapters. We take the Change Rule Effect (CRE) mutation for this example in this the rule's effect will change and we get mutant policy named RPSlist1.xml, let's take the RPSlist.xml's PPS_Student.xml policy's rule effect Permit to Deny [14].

After getting the mutant policy we perform the analysis by OASIS XACML tool for analysis, which gives the output in the form of response. We provide the policy and the request to the OASIS XACML tool and it gives corresponding response.

If we observe the Figure 4.4 and 4.5, the response gets by original policy is *Permit*, whereas the response of mutation policy is *Deny*. If we compare both the

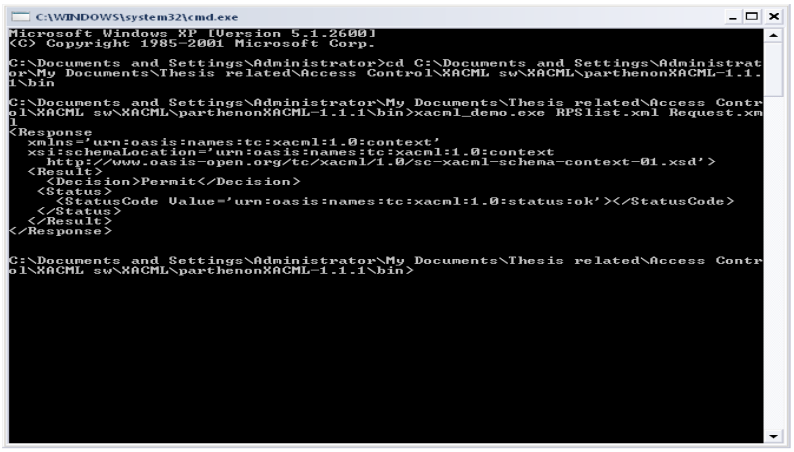

Figure 4.4: Response on Original Policy

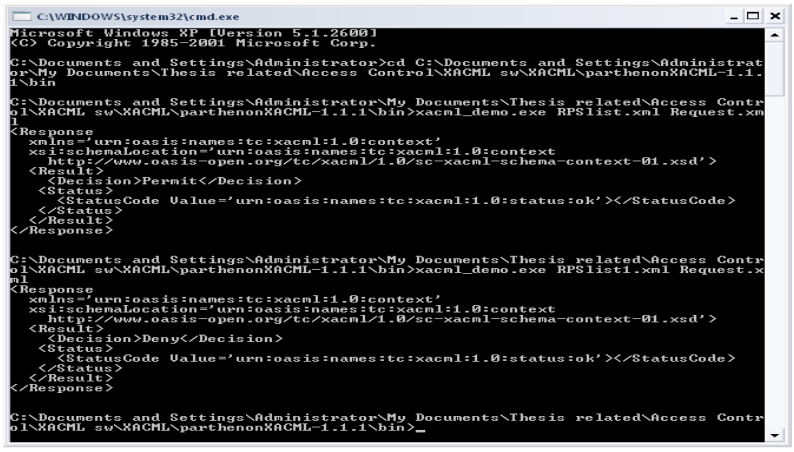

Figure 4.5: Response on Mutant Policy

responses are not same, in this way we can say that the mutant is killed.

4.5 Conclusion

In this chapter, we have seen the different sample policies which are used in the experiment and actual implementation of ACPC with the help of one example. The working of Margrave tool and the working of OASIS XACML tool which is clearly mention and illustrated in the form of figures.

Chapter 5

Result and Discussion

Introduction

Mutation Operator Classes

Metrics

Fault Detection Capability Comparision

Classified Mutation Operators

Metrics

Conclusion

Chapter 5

Result and Discussion

5.1 Introduction

In this chapter, we describe our result in the form of tables and graphs. Before that we classify the mutation operators into different classes as mentioned in [6] and we define some metrics [14] which used in the experiment. Every result contains the table, followed by graphs and discussion. In the last we conclude our result by showing the performance of our framework.

5.2 Mutation operator classes

The mutation operators can be classified based on the policy element on which the mutation operation is performed. They can be classified as,

Policy Set Mutation Operators : These represent the mutation operations that can be done at the policy set level. The various mutation operators defined for this is shown in Table 5.1. They are, policy set target true mutant in which the policy set target is removed so that it is always true, policy set target false mutant in which the target value is changed such that it is always evaluated to false and change in policy combining algorithm mutant in which mutants are created for each policy combining algorithm like permit overrides, deny overrides and first applicable.

Policy Mutation Operators : These represent the mutation operations that can be done at the policy level. The various mutation operators defined for this is shown in Table 5.2. Those are policy target true and policy target false mutation

Table 5.1: Policy Set mutation operator

ID	Description
PSTT	Policy set Target True
PSTF	Policy Set Target False

Table 5.2: Policy mutation operator

ID	Description
PTT	Policy Target True
PTF	Policy Target False

operators.

These are similar to the policy set operators except that the granularity is at the level of policy rather than policy set.

Rule Mutation Operators: These represent the mutation operations that can be done at the rule level. The various mutation operators defined for this is shown in Table 5.3. They are rule target true, rule target false, rule condition true, rule condition false and change rule effect. The first two rule operators generate mutants by setting the rule targets to be true and false. The condition operators set the condition in each rule to be true or false. The change rule effect changes a rule with an effect of permit to one with an effect of deny and vice versa.

Table 5.3: Rule mutation operator

ID	Description
RTT	Rule Target True
RTF	Rule Target False
RCT	Rule Condition True
RCF	Rule Condition False
CRE	Change Rule Effect

5.3 Metrics

In order to investigate our hypotheses, we need to measure the coverage metrics and the mutant killing rate. The following metrics are measured for each policy under test, each request set, and each mutation operator.

Test count : The test count is the size of the request set or the number of

Table 5.4: Mutant-kill result achieved by both request set

—		**Random Method**		**Change-Impact analysis Method**	
XACML Policy	**# Mutant**	**# Mutant Kill**	**Mutant kill %**	**# Mutant Kill**	**Mutant kill %**
codeA	64	20	31.25%	29	45.31%
codeB	92	33	35.87%	42	45.65%
codeC	112	50	44.64%	53	47.32%
codeD	148	55	37.16%	69	46.62%
default-2	157	10	6.37%	85	54.14%
demo-11	22	16	72.73%	16	72.73%
demo-26	17	09	52.94%	09	52.94%
demo-5	23	17	73.91%	19	82.61%
mod-fedora	157	35	22.29%	82	52.23%
average			41.90%		55.50%

generated tests. For testing access control policies, a test is synonymous with request.

Reduced-test count : Given a policy and the generated set of requests, the reduced test count is the size of the reduced request set based on policy coverage.

Mutant count : Given a policy under test and with the mutant operators the numbers of mutant policies are Mutant count.

Mutant-killing ratio : Given a request set, the policy under test, and the set of generated mutants, the mutant-killing ratio is the number of mutants killed by the request set divided by the total number of mutants.

5.4 Fault Detection Capability Comparison

In Table 5.4 summarizes the two approaches of request generation method by its mutant killing ability which is some how similar as [22]. Column 2 shows the number of mutant policy generated by the Mutation operator Handler it depends upon the policy attributes namely policy set, policy, target and conditions, more the attribute more the mutant policies. Columns 3-4 shows the mutant killed and mutant killed% by request set generated by random method, similarly columns 5-6 shows the mutant killed and mutant killed% by request set generated by change-impact analysis method.

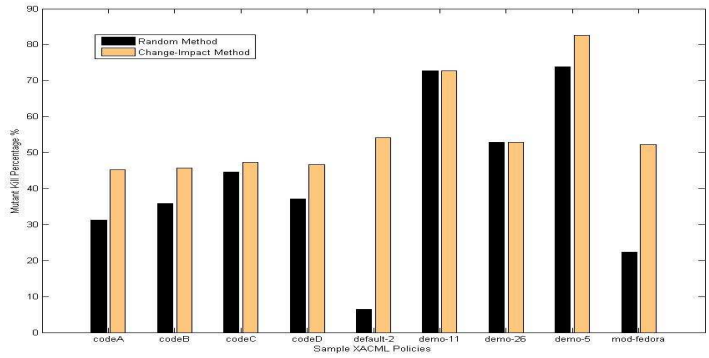

Figure 5.1: Comparison between Random and Change-Impact method

It's observe from the table that, when the mutant kill is higher then the percentage of the mutant kill is also higher. Therefore, we can say that the average mutant killing% of change-impact analysis method is greater than that of random method. Some time the request generated by random method is same as the change-impact analysis method, in the row of demo-11 and demo-26 the killing% is same. But in the case of default-2 there is the huge change, it varies from 6.3% to 54.14%. We can say that our method works well in all the cases and gives good result compare to random method. The graph between sample XACML policies and mutant kill percentage by both methods are shown in Figure 5.1, the difference between random and change-impact method are clearly visible in Figure 5.1.

5.5 Classified Mutantion Operators

As we have classified the mutant operator into three classes [6] Policy Set mutation operator, Policy mutation operator and Rule mutation operator. The main objective of making the class is to check the performance of mutant killing. Table 5.5 shows the individual mutant operator performance in all sample ACML policies and Table 5.6 shows the classified structure is mutant operators. According to each table there is the graph which shows the performance between sample

Table 5.5: Mutant-kill percentage of all XACML policies by individual Mutation Operators

Mutation Operators	**Random method**	**Change-Impact method**
PSTT	64.23%	64.23%
PSTF	61.83%	63.58%
PTT	81.36%	98.14%
PTF	75.6%	96.48%
RTT	72.84%	26.44%
RTF	63.48%	90.14%
RCT	31.23%	12.94%
RCF	77.16%	98.14%
CRE	71.14%	90.88%

Table 5.6: Mutant-kill percentage by Mutation Operators

XACML Policy	**Rule operator kill %**	**Policy operator kill %**	**Policy Set operator kill %**
codeA	89.14%	26.36%	20.43%
codeB	81.08%	27.9%	27.97%
codeC	78.14%	31.24%	32.58%
codeD	80.88%	31.01%	27.97%
default-2	50.94%	30.4%	81.08%
demo-11	88.28%	57.18%	-
demo-26	56.28%	49.6%	-
demo-5	98.14%	67.08%	-
mod-fedora	65.6%	25.6%	22.29%

XACML policies and Mutant killing percentage (Figure 5.1 and Figure 5.3) and between Mutation Operators and Mutation killing percentage (Figure 5.2).

The result of Table 5.6 and corresponding graph shows that Rule mutation operator gives higher mutant killing percentage than Policy Set mutation operator and Rule mutation operator. The conclusion is that the mutation operator that we have chosen the Rule mutation operator works well and with the help of these five mutation operator we can perform the policy testing, which gives the approximate same result as with the all mutant operator, but take less cost and less time.

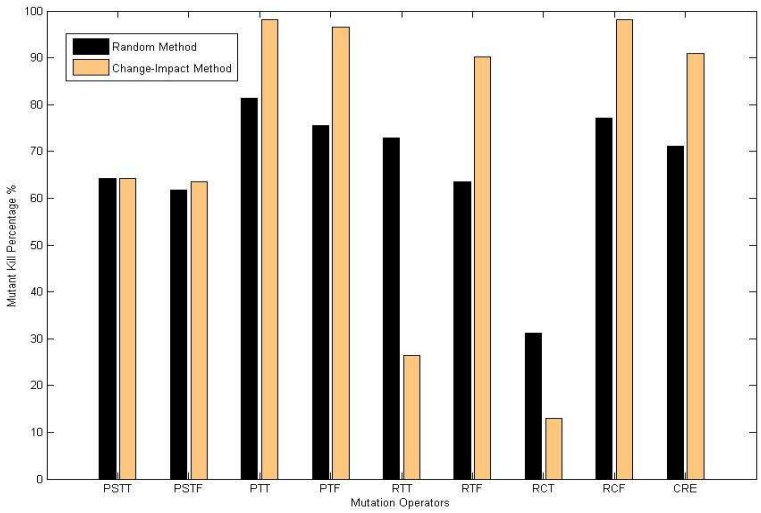

Figure 5.2: Fault detection of all policies by individual mutation operator

5.6 Conclusion

In this chapter we have seen the different result set which compares the result of the framework and random method which has been used so fare. The framework gives better result in the form of greater mutant killing rate with the good quality request set generated by change-impact analysis method.We have classified the mutation operator [6] by which, Rule mutation operator gives better performance then the Policy Set mutation operator and Policy mutation operator. The whole result shown in the form of tables and graphs in this Chapter.

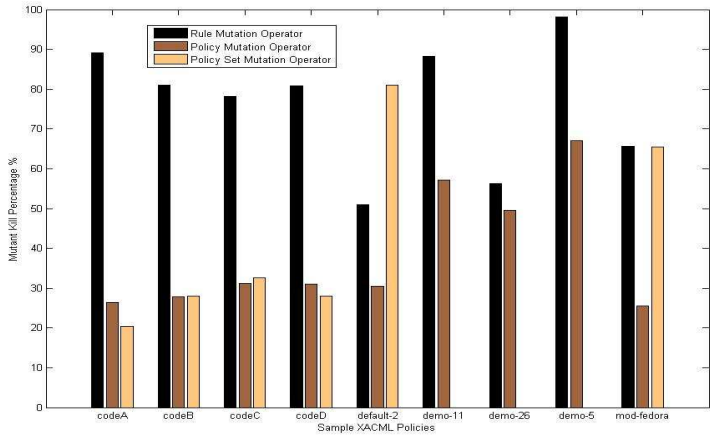

Figure 5.3: Mutant-killing ratio by different class of operators

Conclusion and Future Work

Policy testing is a practical technique for the quality assurance of access control policies. In this thesis, we analysis a method ACPC that uses policy programs for testing access control policies. An automated tool has been developed based on this method. Given a policy, ACPC can generate an optimal number of requests for testing the policy. The advantage of that approach is that, we use existing software testing techniques that are being used for testing different software applications. Also, this approach is general and can be applied for testing most Role based policy specifications even in other languages.

We evaluate the method by testing with nine XACML policies. We perform mutation testing on the policies and the generated request sets and compare the results with the other existing techniques as analysis. The mutant kill percentage is as good as or better than existing techniques in most of the cases. Also, the results indicate that the mutants created by the rule operator have more kill percentage than that achieved by other operators. This shows that the use of the policy program for generating test cases is able to capture fine errors created by mutants. We got up to 98% of mutant killed by the rule mutation operator and this classification [6] gives better performance in terms of cost and time.

In future, we want to develop the better coverage tool based on the some improved criteria. The stronger the coverage criteria, the better will be the quality of the test cases (responses) generated. Also, we will define some new mutation operator so that our framework will become more cost-effective and accurate for testing the access control policies.

Bibliography

[1] E. Lup N. Damianou, N. Dulay and M. Sloman. The ponder policy specification language. *In Proc.International Workshop on Policies for Distributed Systems and Networks*, pages 18–38, 2001.

[2] G.Karjoth C.Powers P.Ashley, S.Hada and M.Schunter. Enterprise privacy authorization language epal. *http://www.w3.org/Submission/EPAL*, 2003.

[3] OASIS eXtensible Access Control Markup Language XACML. *http://www.oasis-open.org/committees/xacml*, 2005.

[4] A. Wool. A quantitative study of firewall configuration errors. *Computer*, 37(6):62–67, 2004.

[5] G. Hughes and T. Bultan. Automated verification of access control policies. *Technical Report Department of Computer Science, University of California*, 2004.

[6] D.Sivasubramanian and Ting Yu. Automated access control policy testing through code generation. *COMPUTER SCIENCE, Raleigh, North Carolina*, 2007.

[7] L. A. Meyerovich K. Fisler, S. Krishnamurthi and M. C. Tschantz. Verification and change-impact analysis of access-control policies. *In Proc. 27th International Conference on Software Engineering*, pages 196–205, 2005.

[8] T. Xie E. Martin and T. Yu. *Defining and measuring policy coverage in testing access control policies.* In Proc. 8th International Conference on Information and Communications Security, December 2006.

[9] R. S. Sandhu and P. Samarati. Access control: Principles and practice. *IEEE Communications*, 32(9):40–48, September 1996.

[10] D.Ferraiolo V.Hu and R. Kuhn. Assessment of access control systems. *NIS-TIR*, September 2006.

[11] Messaoud Benantar. *Access Control Systems: security, Identity Management and TTrust Models*. Springer, 2008.

[12] H. L. Feinstein R. S. Sandhu, E. J. Coyne and C. E. Youman. Role-based access control models. *IEEE Computer*, 29(2):38–47, February 1996.

[13] R. Sandhu S. Osborn and Q. Munawer. Configuring role-based access control to enforce mandatory and discretionary access control policies. *ACM Transactions on Information and System Security*, 3(2):85–106, May 2000.

[14] Evan Martin and Tao Xie. *Automated Test Generation for Access Control Policies via Change-Impact Analysis*. May 2007.

[15] R. A. DeMillo and A. J. Offutt. Constraint-based automatic test data generation. *IEEE Transaction Software Engineering*, 17(9):900–910, 1991.

[16] L. J. morell. A theory of fault-based testing. *IEEE Transaction Software Engineering*, 16(8):844–857, August 1990.

[17] Daniel Jackson. *Software Abstractions Logic, Language and Analysis*. The MIT Press, 2006.

[18] H.Hamed A. El-Atawy, K. brahim and E. Al-Shaer. Policy segmentation for intelligent firewall testing. *In 1st IEEE ICNP Workshop on Secure Network Protocols*, pages 67–72, November 2005.

[19] Evan Martin and Tao Xie. Toward systematic testing of access control policies. *Proceedings of MASPLAS06, Mid-Atlantic Student Workshop on Programming Languages and Systems Rutgers University*, April 2006.

[20] G. M. Kapfhammer and M. L. Soffa. A family of test adequacy criteria for database-driven applications. *In Proc. 9th European Software Engineering Conference held jointly with 11th ACM SIGSOFT International Symposium on Foundations of Software Engineering*, pages 98–107, Septemper 2003.

[21] PLT scheme with Drscheme package. *http://www.plt-scheme.org/.*

[22] Evan Martin and Tao Xie. A fault model and mutation testing of access control policies. *In Proceedings of the 16th International Conference on World Wide Web (WWW 2007)*, pages 667–676, May 2007.

[23] E. Sirer and K. Wang. An access control language for web services. *In Proc. 7th ACM Symposium on Access Control Models and Technologies, Monterey, CA*, pages 23–30, June 2002.

[24] X. Zhang T. Jaeger and A. Edwards. Policy management using access control spaces. *ACM Transactions on Information and System Security (TISSEC)*, 6(3):327–364, August 2003.

[25] Margrave's API version 2. *www.cs.brown.edu/research/plt/software/margrave/.*

[26] F.Somenzi.CUDD. The cudd decision diagram package. *http://vlsi.colorado.edu/ fabio/CUDD/.*

[27] R. Majumdar D. Beyer, A. J. Chlipala. Generating test from counterexamples. *In Proc. 26th International Conference on Software Engineering*, pages 326–335, 2004.

[28] A. J. Offutt R. Geist and F. c. Harris. Estimation and enhancement of real-time software reliability through mutation analysis. *IEEE Transactions on Computers*, 41(5):55–558, May 1992.

[29] J. Offutt and R. H. Untch. Mutation 2000: Uniting the orthogonal. *In Mutation 2000: Mutation Testing in the Twentieth and the Twenty First Centuries, San Jose, CA*, pages 45–55, October 2000.

[30] M. Hennessy and J. F. Power. An analysis of rule coverage as a criterion in generating minimal test suites for grammar-based software. *In Proc. 20th IEEE/ACM International Conference on Automated Software Engineering*, pages 104–113, November 2005.

Appendix A

Sample XACML policy

```
<?xml version="1.0" encoding="UTF-8" ?>
<PolicySet xmlns="urn:oasis:names:tc:xacml:1.0:policy"
   xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
   xsi:schemaLocation="urn:oasis:names:tc:xacml:1.0:policy
   /pro/xacml/summer2004/xacml/schema/cs-xacml-schema-policy-01.xsd"
   PolicySetId="RPSlist" PolicyCombiningAlgId="urn:oasis:names:tc:xacml:1.0:policy-
   combining-algorithm:permit-overrides">
 <Target />
 <PolicySetIdReference>RPS_Faculty</PolicySetIdReference>
 <PolicySetIdReference>RPS_Student</PolicySetIdReference>
</PolicySet>
```

Figure A.1: RPSlist.xml

```
<?xml version="1.0" encoding="UTF-8" ?>
<PolicySet xmlns="urn:oasis:names:tc:xacml:1.0:policy"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xsi:schemaLocation="urn:oasis:names:tc:xacml:1.0:policy
  /pro/xacml/summer2004/xacml/schema/cs-xacml-schema-policy-01.xsd"
  PolicySetId="RPS_Student"
  PolicyCombiningAlgId="urn:oasis:names:tc:xacml:1.0:policy-combining-
  algorithm:permit-overrides">
<Target>
<Subjects>
<Subject>
<SubjectMatch MatchId="urn:oasis:names:tc:xacml:1.0:function:string-equal">
  <AttributeValue
  DataType="http://www.w3.org/2001/XMLSchema#string">Student</AttributeValue>
<SubjectAttributeDesignator AttributeId="role"
  DataType="http://www.w3.org/2001/XMLSchema#string" />
</SubjectMatch>
</Subject>
</Subjects>
</Target>
<PolicySetIdReference>PPS_Student</PolicySetIdReference>
</PolicySet>
```

Figure A.2: RPS_Student.xml

```
<?xml version="1.0" encoding="UTF-8" ?>
<PolicySet xmlns="urn:oasis:names:tc:xacml:1.0:policy" xmlns:xsi="http://
www.w3.org/2001/XMLSchema-instance"
xsi:schemaLocation="urn:oasis:names:tc:xacml:1.0:policy /pro/xacml/summer2004/
xacml/schema/cs-xacml-schema-policy-01.xsd" PolicySetId="RPS_Faculty"
PolicyCombiningAlgId="urn:oasis:names:tc:xacml:1.0:policy-combining-
algorithm:permit-overrides">
<Target>
<Subjects>
<Subject>
<SubjectMatch MatchId="urn:oasis:names:tc:xacml:1.0:function:string-equal">
 <AttributeValue DataType="http://www.w3.org/2001/XMLSchema#string">Faculty</
AttributeValue>
 <SubjectAttributeDesignator AttributeId="role" DataType="http://www.w3.org/2001/
XMLSchema#string" />
 </SubjectMatch>
 </Subject>
 </Subjects>
 </Target>
 <PolicySetIdReference>PPS_Faculty</PolicySetIdReference>
 </PolicySet>
```

Figure A.3: RPS_Faculty.xml

```
<?xml version="1.0" encoding="UTF-8" ?>
<PolicySet xmlns="urn:oasis:names:tc:xacml:1.0:policy" xmlns:xsi="http://www.w3.org/2001/
XMLSchema-instance" xsi:schemaLocation="urn:oasis:names:tc:xacml:1.0:policy /pro/xacml/
summer2004/xacml/schema/cs-xacml-schema-policy-01.xsd" PolicySetId="PPS_Student"
PolicyCombiningAlgId="urn:oasis:names:tc:xacml:1.0:policy-combining-algorithm:permit-
overrides">
  <Target />
<Policy PolicyId="gradedp" RuleCombiningAlgId="urn:oasis:names:tc:xacml:1.0:rule-combining-
algorithm:permit-overrides">
  <Target />
<Rule RuleId="graded-View-grades" Effect="Permit">
<Target>
<Resources>
<Resource>
<ResourceMatch MatchId="urn:oasis:names:tc:xacml:1.0:function:string-equal">
  <AttributeValue DataType="http://www.w3.org/2001/XMLSchema#string">ExternalGrades</
AttributeValue>
  <ResourceAttributeDesignator AttributeId="resource-class" DataType="http://www.w3.org/2001/
XMLSchema#string" />
  </ResourceMatch>
  </Resource>
  </Resources>
<Actions>
<Action>
<ActionMatch MatchId="urn:oasis:names:tc:xacml:1.0:function:string-equal">
  <AttributeValue DataType="http://www.w3.org/2001/XMLSchema#string">Receive</
AttributeValue>
  <ActionAttributeDesignator AttributeId="command" DataType="http://www.w3.org/2001/
XMLSchema#string" />
  </ActionMatch>
  </Action>
  </Actions>
  </Target>
  </Rule>
  </Policy>
  </PolicySet>
```

Figure A.4: PPS_Student.xml

```
<?xml version="1.0" encoding="UTF-8" ?>
<PolicySet xmlns="urn:oasis:names:tc:xacml:1.0:policy" xmlns:xsi="http://www.w3.org/2001/
XMLSchema-instance" xsi:schemaLocation="urn:oasis:names:tc:xacml:1.0:policy /pro/xacml/
summer2004/xacml/schema/cs-xacml-schema-policy-01.xsd" PolicySetId="PPS_Faculty"
PolicyCombiningAlgId="urn:oasis:names:tc:xacml:1.0:policy-combining-algorithm:permit-
overrides">
 <Target /> <Policy PolicyId="Assignp" RuleCombiningAlgId="urn:oasis:names:tc:xacml:1.0:rule-
combining-algorithm:permit-overrides">
 <Target />
<Rule RuleId="gradeAsgn-Assign-grades" Effect="Permit">
<Target><Resources><Resource>
<ResourceMatch MatchId="urn:oasis:names:tc:xacml:1.0:function:string-equal">
 <AttributeValue DataType="http://www.w3.org/2001/XMLSchema#string">ExternalGrades</
AttributeValue>
 <ResourceAttributeDesignator AttributeId="resource-class" DataType="http://www.w3.org/2001/
XMLSchema#string" />
 </ResourceMatch> </Resource>
<Resource><ResourceMatch MatchId="urn:oasis:names:tc:xacml:1.0:function:string-equal">
<AttributeValue DataType="http://www.w3.org/2001/XMLSchema#string">InternalGrades</
AttributeValue>
 <ResourceAttributeDesignator AttributeId="resource-class" DataType="http://www.w3.org/2001/
XMLSchema#string" />
 </ResourceMatch>
 </Resource> </Resources>
<Actions><Action>
<ActionMatch MatchId="urn:oasis:names:tc:xacml:1.0:function:string-equal">
 <AttributeValue DataType="http://www.w3.org/2001/XMLSchema#string">Assign</
AttributeValue>
 <ActionAttributeDesignator AttributeId="command" DataType="http://www.w3.org/2001/
XMLSchema#string" />
 </ActionMatch> </Action><Action>
<ActionMatch MatchId="urn:oasis:names:tc:xacml:1.0:function:string-equal">
 <AttributeValue DataType="http://www.w3.org/2001/XMLSchema#string">View</
AttributeValue>
 <ActionAttributeDesignator AttributeId="command" DataType="http://www.w3.org/2001/
XMLSchema#string" />
</ActionMatch>
</Action>
</Actions>
</Target>
</Rule>
</Policy>
</PolicySet
```

Figure A.5: PPS_Facilty.xml